# "A Tale of Two Provers":

## A Comparison of Dependent Haskell and F*

Xinyue Zhang
Advised by Richard A. Eisenberg
March, 2018

**Abstract**

This is the abstract.

still need to update this.

Submitted in Partial fulfillment of the requirement for Bachelor of Arts in Computer Science at Bryn Mawr College

**To my Dearest Grandpa Zhaomin,**

who has supported me in his own way toward this bachelor.

# Acknowledgements

My sincere gratitude goes first to my advisor, Professor Richard Eisenberg, whose unwavering passion and patient guidance have kept me constantly engaged in my research. I am very grateful to Professor Dianna Xu and Professor Deepak Kumar, whose insight, advice and encouragement have been invaluable to me over these four years. I sincerely appreciate the Computer Science departments at Bryn Mawr and Haverford Colleges, and my special thanks goes to Kris Micinski.

I would like to thank the F* developers, Nikhil Swamy and Tahina Ramananandro for their timely advice. I would like to thank all members of the PL Club at UPenn, especially Yishuai Li for his support along the way. Special shout out to Rachel Xu, my fellow researcher for one and a half year up to now.

I would also like to thank my reviewers: Stephanie Cao, Nora Broderick, Trista Cao and Kara Breeden. I am very grateful for their valuable feedback about my drafts.

I would like to express my sincere gratitude to my parents. Without their unconditional love, I wouldn't be able to achieve thus far. Also I am wholeheartedly thankful for my friends, especially Jordan Henck and Calla Carter, whose selfless companionship is more than I could ever ask.

Finally, to my dearest grandpa Zhaomin, no words can express how much I love you and miss you. I couldn't imagine how happy and proud you would be if you were to attend my graduation.

# Contents

# 1 Introduction and Motivation

> *"Testing can show the presence of errors, but not their absence."*
>
> – Edsger Wybe Dijkstra

Programming languages are powerful tools used to design and build complex computer software in order to perform various tasks. Yet complex software programs often have bugs that prevent them from functioning as expected. Almost everything these days is dependent on software, from smart devices to vehicles, and to national defense systems. Most of the time bugs cause small problems such as an application crash. However, in many cases, tiny errors in a program cause catastrophic if not life threatening consequences such as a financial crisis or a plane crash. Therefore, various research must be done to increase our confidence on the correctness of a program.

Through systematic testing, it is possible to find some incorrect program behaviors, assuming that it is easy or even possible to check the correctness of a program output. Yet exhaustive testing like this is often limited to a given set of test cases, making it not suitable as a general solution. As Edsger Dijkstra stated, "testing can show the presence of errors, but not their absence", testing cannot guarantee the correctness of a program [4]. Other options for error checking exist and formal verification is the most widely used technique.

this is a specific, testable claim. Is it true? I doubt it.

Formal verification uses formal methods, such as logic, to ensure the correctness of software with respect to a defined or inferred specification. We say software is correct if it meets its specification, which defines all its intended behaviors. There are various techniques towards formal verification, each with its own advantages and disadvantages. For example, model checking and run-time monitoring are two lightweight formal methods in use today. Yet, type system design is by far the most widely-adopted and well-founded method [16]. A type system is a mechanism that enforces rules, known as the type rules, on programs to help reason about them. A type classifies values into categories that share certain properties and a type rule consists of principals that enforce conditions on the types to prevent specified errors. For example, we say that `true` and `false` are both of type `Boolean`. Since the type rule on booleans doesn't support the addition operation(`+`), we can catch invalid operations such as `true + false`. In general, a type system can be applied to programming languages to help verify that there are no bugs in the software written. Well designed type systems can help make sure, at compile time, that all programs written perform as intended. Therefore, the desire to design a stronger type system to allow more programs to be typed is a major focus in the programming languages research.

where? provide citations/links

what does this mean here?

this sentence doesn't say much.

& expressions? ie

not no bugs. Type systems don't generally catch all bugs.

Programming languages like Coq [5], Agda [15], and Dafny [13], commonly known as proof assistants, have been studied for a long time and are relatively well understood. They laid down a solid foundation for program verification, but are limited when applied to general-purpose applications in real life due to their exorbitant development requirements. However there are various industrial-strength functional programming languages that are comparably easy to read, program and maintain. With their resemblance to mathematical functions, these functional programming languages also provide a good ba-

what do you mean by this?

what do you mean here? expand

as easy as Coq? that's what "comparably" means here.

sis for supporting verifications through an enhanced type system design. Researchers on functional programming languages, like Haskell and ML, gradually start to support verifications with dependent types, resulting in the design of verification-oriented Dependent Haskell and F*. Each presents a unique and effective program verification technique.

Dependent Haskell and F* are similar in that they both aim towards a verification-based yet general-purpose programming language. They are, however, noticeably different in their unique approaches to verify potential divergent functions. Dependent Haskell doesn't require termination checking, whereas F* categorizes divergent functions into a special `Dv` effect (See section 2.5.4 for details). Most dependently typed verifiers require compile-time termination checking to ensure the correctness of proofs. However, as Alan Turing proved via the Halting Problem, given an arbitrary program and its inputs, checking whether this program will terminate or loop forever is an undecidable problem. Yet functions that diverge or hard to prove termination are very common in practical programs. Therefore, we are interested in exploring how to better express and reason about them.

In this undergraduate thesis, we aim to introduce two verification-oriented programming languages Dependent Haskell and F*, present their unique type system designs with concrete examples, and give a detailed comparison from a combination of theoretical and practical standpoints. We expect our research to setup a solid foundation for potential collaborations across the two communities in the future, and to possibly propose promising research directions in formal verifications for the general programming languages community.

# 2 Background

## 2.1 From Imperative to functional programming

Most programmers come from a Java or C/C++ background [2]. These languages are designed to primarily support imperative or procedural programming. In the imperative style of programming, a developer writes code that details each step for computers to execute. These imperative programs often have side effects that modify program inputs or global variables. Any function modifying global states in execution is known as stateful. For example, global variables can be modified in any function in scope. Functional programming, as its name suggests, models a problem as a set of functions to be executed, detailing the inputs and outputs for each function. Some common functional programming languages are Haskell, ML, Lisp and Scheme. Similar to the mathematical functions introduced in high school algebra, functions in these languages are abstracted to operate on any structured data types, such as boolean, string, list, etc. Functional programs are usually executed by evaluating expressions and typically avoid mutating program states. Functional programming languages can be typed, as in Haskell and ML, or untyped as in Lisp and Scheme. A language is said to be typed if it has a static type system where all values must be classified into types and checked with the system at compile time.

One major difference between the imperative and functional style of programming is

how they handle program iterations. Imperative programming languages usually support both loops and recursion, however functional programming languages depend highly on recursion and higher-order functions, like `map`, `filter` etc. For example, if one wants to double every element of the list `[1,5,2,4,3]`, one can implement in Java, an imperative programming language as shown below

```
public void double(int[] arr) {
    for(int i = 0; i < arr.length; i++) {
      arr[i] = 2 * arr[i];
    }
}
```

Notice that in this `double` example, the input list `[1,5,2,4,3]` is directly modified after the execution of the function to the result `[2,10,4,8,6]`.

On the other hand, the same operation can be defined in Haskell in functional style as seen in the code below,

```
double :: [Int] -> [Int]
double []     = []
double (x:xs) = (2 * x) : double xs
```

In typed functional programming languages, programmers often declare type signatures of a function on the first line and give actual implementations of it on the following lines. In the `double` example above, the input is specified as `[Int]`, denoting a list of integers, and output also as `[Int]`. The entire type definition `[Int] -> [Int]` after the double colon belongs to the type level. Specifically, when applying `double` to the input list as in `double [1,5,2,4,3]`, we multiply the first element in the input list by 2 and prepend the result to the recursive call on the remaining list until we reach the empty list.

## 2.2 Program Specification with Dependent and Refinement Types

The starting point of any program verification is to specify its properties. There are three main ways to specify a program, through dependent type and refinement type. A dependent type is a type whose definition is predicated upon a dynamic value. As such, dependently-typed programming languages enable programmers to express more detailed specifications with types and to perform more powerful verifications through type checking. On the other hand, a refinement type is a type that satisfies a given logical predicates. It helps reduce code redundancy and makes specifications more intuitive.

To illustrate these definitions in more detail, consider the example of defining a list. In imperative programming languages like Java or C/C++, elements in a list must all share a single type, such as the boolean type. Take Java as an example, an array of length 5 will be constructed as `boolean[5]`. To specify the list carrying its own length in verification-oriented programming languages, one can predicate the type of this list of booleans $v$ on its length 5, as in the type `Vec 5 Bool`, with the defined length-indexed `Vec` datatype. The list $v$'s type `Vec 5 Bool` is said to be dependently-typed as it is indexed

but with bodean[5] in the same fg, it's really unclear what's different betw/ Java and Haskell.

on the number 5 of another type `Nat` (the type `Nat` represents the natural numbers, defined to be non-negative integers). Similarly, this list of booleans of length 5 can be specified through logical predicates using refinement types. Recall that refinement types specify a logical formula that the types have to additionally satisfy. So one must first define a function `len` calculating the length of a list and then apply it in the predicate to produce the refinement type `v:List Bool{len v = 5}`. The logical formula `len v = 5` is called the predicate of the refinement type, and `list` is the built-in data type in F*. Both dependent types and refinement types are commonly applied in program verifications. Dependent Haskell only supports dependent types, but F* supports both dependent types and refinement types. We will compare them in detail in our result section.

refinement types can be implemented in terms of dependent types.

## 2.3 Dependent Haskell

### 2.3.1 Introduction to Haskell

Haskell is a strongly typed purely functional programming language.

As a pure functional programming language, each function in Haskell only takes in some inputs and returns an output after some operations, without modifying any unspecified program states. Unlike many functions (which should really be called procedures) in an impure language, functions in Haskell don't mutate variables or handle errors. They also don't read or write to standard inputs or outputs, nor do they connect to sockets to communicate with the outside world. This seemingly limiting feature ensures that each execution of the function always returns exactly the same result, which allows formal proofs on functions and also composing functions to perform more complex operations.

Haskell is a typed language, where every variable's type is determined at compile time and is checked before performing any operations. In addition, Haskell's type system enforces strict typing rules on functions, making Haskell a strongly-typed language. In strongly-typed languages, every variable type is predefined and requires explicit casts when used differently. By this definition, Java is strongly typed since it rejects the mixed-typed operations like `2 * "Haskell"` but C is weakly typed since one can perform arithmetic on pointers to bypass the type system.

not Java does not always require casts. but these need casts!

Finally, Haskell is lazy, since every Haskell operation is lazily evaluated. This means that the final result of an operation is not computed unless required. Suppose we want to quadruple the list `l = [1,2,3,4]` using our `double` function by calling `r = double (double [1,2,3,4])`. In non-lazy languages, the two `double` expressions will be evaluated right away to produce the resulting list `[4,8,12,16]` and assign it to `r`. However, in lazily evaluated languages, evaluations are deferred until results are needed by other computations. For example, the evaluation will occur when the programmer wants to get the first element in the resulting quadrupled list. Specifically, the first `double` will request a result from the second "inner" `double`. Then, the second "inner" `double` will be executed to produce the doubled list. Finally the outer `double` takes the resulting doubled list as input, outputs the quadrupled list, and returns its first element. The biggest advantage of a lazy language is to improve code modularity without sacrificing code efficiency.

This is not how laziness works in this example. Each element is quadrupled separately. Put an 'undefined' in the list to experiment!

### 2.3.2 Dependently-Typed Haskell

For a long time, Haskell researchers have striven to add dependent types in Haskell, as dependent types introduce much more precise expressions of program specifications. Although Haskell doesn't yet support dependent types in full, several extensions to its current type system have made exceptional progress.

In 2012, Eisenberg and Weirich introduced the `singletons` library that simulates dependently-typed programming in Haskell. Through dependent types, programmers can reason about programs through computations on the indexed types. For example, when concatenating two lists of booleans, each of length `m` and `n`, the output list can be specified to have type `Vec (Plus m n) Bool`, indexed on the type variable `Plus m n` representing the sum of the two lengths of the original lists. However, Haskell enforces two separate scopes in programming functions – the terms executed at run time and the types checked at compile time. Functions in these two scopes cannot be used interchangeably, so singleton types are necessary to bridge the gap.

Beyond Singletons, Gundry and Eisenberg both conducted research on dependently-typed Haskell using GADTs and type families (Detailed definitions and examples are provided in section 2.4 Program Verification) [11, 7]. Eisenberg, extending Gundry's work, brings Haskell to a full-spectrum dependently typed language. Eisenberg's work, now partially implemented in the Glasgow Haskell Compiler (GHC) 8.0, finally provides a backward compatible type system design to support real dependent types in Haskell [7].

## 2.4 Program Verification in Dependent Haskell

After a brief overview of Haskell and Dependent Haskell, this section focuses on Dependent Haskell's verification techniques. The section is structured in a tutorial manner, where concepts are introduced sequentially and gradually build up to two program verification examples as `insert` and `concat`.

### 2.4.1 Algebraic Data Types

Type system based program verification is mainly focused on types. Beyond the primitive types, `Int`, `Bool` etc, which are defined in the standard library, one can also define one's own types. Similar to the C structs, new types can be introduced through the `data` keyword.

For example, a list of elements of type `a` is defined recursively in Haskell as:

```
data List a = Nil | Cons a (List a)
```

The `List` type has two value constructors `Nil` and `Cons`. Each constructor specifies a value that the type could have. Specifically, `Nil` represents the empty list and `Cons`, sometimes written as `:>`, combines one more element to an existing list to form a new list. For example, suppose we have a list of integers `l1 = [2,3,4,5]`, the expression `Cons 1 l1` represents the new list `l2 = [1,2,3,4,5]`. The element `1` that is add into

the list `l1` is called the head of `l2`, and the rest of the list is called the tail of `l2`. The symbol | represents logical or, so the `List` type defined above can have a value either an empty list or a list of elements of type `a`. The type `a` could represent any type such as `Int`, `Bool` etc, as long as all the `a`s are matched consistently. In general, any type defined with value constructors is called an algebraic data type (ADT).

Similarly, one can formally define the natural numbers recursively in Dependent Haskell as:

```
data Nat = Zero | Succ Nat
```

where `Zero` represents the smallest natural number 0, and `Succ` represents all the remaining nonzero natural numbers. For example, 1 is expressed as `Succ Zero` and 2 is expressed as `Succ (Succ Zero)` etc.

### 2.4.2 Generalized Algebraic Data Types

It follows that, by using dependent types, one can encode the length of a list into its type as a natural number, by parameterizing the list by its length and its element type. Below we introduce the dependently typed vector definition in Dependent Haskell:

```
data Vec :: Nat -> Type -> Type where
  Nil  :: Vec Zero a
  (:>) :: a -> Vec n a -> Vec (Succ n) a
```

this uses a very different syntax than what you introduced above.

Instead of a simple listing of value constructors, the definition of `Vec` extends that of `List` by explicitly clarifying the type signature for each constructor. The `Vec` datatype is defined to take in a natural number to specify the length of a vector and the type of each element in the vector, resulting in a new type of kind `Type`. Notice that the `Vec` type has the same two value constructors `Nil` and `Cons` (represented as the symbol `(:>)`) as in the `List` type definition above. The constructor `Nil` has type `Vec Zero a`, meaning that it's an empty list of length `0` and each element has type `a`. Again, this type `a` can represent any type in the Haskell type system. The second constructor `Cons` is recursively defined. It adds a new element of type `a` to an existing list of length `n` and result in a new list of length `n+1`. The above definition on `Vec` is called a Generalized Algebraic Data Type, or a GADT. Specifically, pattern matching on `Nil` gives us type `Vec Zero a` and pattern matching on `x :> xs` gives us type `Vec (Succ n) a`. In general, a GADT allows programmers to get specific information about types thorough patterning matching on each value constructor.

is it Cons or :> pick one.

unhelpful

### 2.4.3 A Simple Verification Example

With the introduced concepts as parameterized dependently-typed GADT `Vec`, we begin with a simple example on how program verification works in Haskell.

Specifically, one can define the `insert` function in Dependent Haskell to insert an element to a sorted list.

```
insert :: Ord a => a -> Vec n a -> Vec (Succ n) a
insert elm Nil       = elm :> Nil
insert elm (x :> xs) = if elm <= x then elm :> x :> xs else x :> (insert elm xs)
```

Elements in the list of type `a` are all comparable, since we define `a` to belong to the `Ord` type class for all types that have an ordering. A type class can be understood as an interface that categorizes a group of behaviors. In the `insert` example above, we define the generic type `a` to be a part of the `Ord` type class since we want to perform comparison as in `elm <= x` among terms of type `a`. Also, the above implementation uses GADT pattern matching to classify which case to apply. Specifically, `Nil` is to pattern match the original input list with the empty list, and `(x :> xs)` is to pattern match the original input list with at least an element. The above insert algorithm may be understood imperatively as follows: the element `elm` is inserted to the list if the list input is originally empty. Otherwise, as the list contains ordered elements, one may compare the value of `elm` with the smallest element `x` in the original nonempty list, and either recursively insert the new element into the original list or directly prepend the element to the front of the list. Notice that when the above `insert` algorithm is repeatedly applied, the eventual output list will be sorted.

Similar to writing test cases, program verifications also focus on the specification of each program. For insert, the length of the list is incremented by `1` every time one inserts a new element. Therefore, the return type for function `insert` can be defined as `Vec (Succ n) a`, encoding the length incrementation in the returned GADT.

After encoding the verification condition in the type level, we will explain step by step how the actual verification is carried out. First, we verify that adding an element to an empty list results in an list of length `1`.

```
insert elm Nil = elm :> Nil
```

This is obvious since we can directly apply our definition of the cons operator `:>` with type `a -> Vec Zero a -> Vec (Succ Zero) a`. So the result list `ele :> Nil` has type `Vec (Succ Zero) a`, representing a list of length `1` with elements of type `a`.

Second, we verify that adding an element to a list of length `n > 0` results in a list of length `n + 1`.

```
insert elm (x :> xs) = if elm <= x then elm :> x :> xs else x :> (insert elm xs)
```

Specifically, following the definition in the last line of the above `insert` function, we check if the element `elm` is less than the first(i.e. smallest) element `x` in the sorted input list and separately consider the two cases as follows. Notice that in both cases, the list `x :> xs` has type `Vec n a` from pattern matching.

***Case 1:*** *`elm <= x`*

Let us first consider the case when the inserted element `elm` is less than or equal to the smallest element `x` in the sorted input list. From the definition, we obtain the result `elm :> x :> xs` where `x :> xs` has type `Vec n a`. We can directly follow the definition of the cons operator `:>` and conclude that the result list `elm :> (x :> xs)` has type `Vec`

`(Succ n) a`, with one more element than the input list.

***Case 2:*** *`elm > x`*

Finally, we consider the last case when the inserted element `elm` is greater than the smallest element `x` in the sorted input list. We know from pattern matching that the input list `x :> xs` has `n` elements. Consider the tail of the list, i.e. `xs`. If we let `xs` have length `m`, then we know `n = m + 1`. Thus, we want to check that if the input list has type `Vec (Succ m) a`, then the resulting list `x :> (insert elm xs)` has type `Vec (Succ (Succ m)) a`. Since `xs` has type `Vec m a`, by induction hypothesis, we can conclude that `insert elm xs` has type `Vec (Succ m) a`. As a result from the definition of the cons operator, the result list has type `Vec (Succ (Succ m)) a`, which is what we want to verify.

### 2.4.4 Type Family

To perform some more involved computations in type level, such as addition or multiplication, Dependent Haskell introduce what's known as a type family. For example when concatenating two `Vec`s, one needs to verify the specification such that the resulting `Vec` has length the sum of the two concatenated `Vec`s.

Type family is a function defined at type level that returns a type. It gives us an extended ability to compute over types and is the core of type-level computation. For example, to add two type-level `Nat`s together, we define the type family in Dependent Haskell as:

```
type family Plus (a :: Nat) (b :: Nat) :: Nat where
  Plus Zero b      = b
  Plus (Succ a') b = Succ (Plus a' b)
```

The implementation of `Plus` is exactly the same as the ordinary `(+)` implementation in the term level, except that `Plus` is in the type level. Again, we apply pattern matching on the first summand of plus. If the first summand `a` is `0`, then we define our base case such that any sum of `0` and another natural number `b` is just the other natural number `b`, or in short `0 + b = b` for all natural number `b`. Otherwise, if the first summand is a non-zero natural number, represented as `Succ a` where `a'` is the natural number one less than `a`, then one recursively applies `Plus` on the smaller natural number `a'` and the other natural number `b` to obtain the sum and add one to obtain the result. Equivalently, in algebraic equations, if `a = 1 + a'` then the operation is essentially `a + b = (1 + a') + b = 1 + (a' + b)` from associativity of plus.

### 2.4.5 Program Verification with Dependent Types and Type Families

Finally, applying both dependent types and type families, one can verify the operation of concatenating two lists. The operation `concat` can be recursively implemented and verified in Dependent Haskell as:

```
concat :: Vec m a -> Vec n a -> Vec (Plus m n) a
concat Nil l2       = l2
concat (x :> xs) l2 = x :> (concat xs l2)
```

The implementation is similar as before, applying pattern matching on the first parameter and recurse on the smaller list. After concatenating two lists, the resulting list is expected to have length the sum of the lengths of the two original lists, as specified in the dependent type `Vec (Plus m n) a` with the `Plus` type family definition.

Then, we discuss the verification in detail for the following two cases. First we want to show that concatenating an empty list with any other list `l2` results in a list of length the sum of zero and the length of `l2`.

```
concat Nil l2 = l2
```

This is immediate from the definition of the `Plus` type family, as `Plus Zero n = n`. Thus, the result list `l2` has type `Vec n a = Vec (Plus Zero n) a`.

Second, we want to show that concatenating any nonempty list `x :> xs` with another list `l2` results in a list of length the sum of the lengths of the two input lists.

```
concat (x :> xs) l2 = x :> (concat xs l2)
```

From pattern matching, we know that the first input list `x :> xs` have type `Vec m a` and the second input list `l2` has type `Vec n a`. Let the tail of the first input list, i.e. `xs` to have length `m'`. Then we know that `m = m' + 1` and so `x :> xs` has type `Vec m a = Vec (Succ m') a`. Given the type signature of `concat`, we then want to check if the result list `x :> (concat xs l2)` has type `Vec (Plus (Succ m') n) a`. From the inductive hypothesis, we can show that `concat xs l2` has type `Vec (Plus m' n) a`. It follows that, from the definition of the cons operator, the list `x :> (concat xs l2)` has type `Vec (Succ (Plus m' n) a`. Finally, applying the second equation in the definition of the `Plus` type family on `Succ`, we can conclude that the result list has type `Vec (Succ (Plus m' n) a = Vec (Plus (Succ m') n) a`, as we expected.

## 2.5 FStar

F*, pronounced FStar, is another functional programming language aimed at program verification [17]. It is developed at Microsoft Research, MSR-Inria and Inria and follows a similar syntax to the languages in the ML family [17]. In 2016, Swamy et al. introduced the current, completely redesigned version of F*. The new F* is a general-purpose, verification-oriented and effectful programming language. F* is said to be general-purpose as it is expressive enough to support programming in various practical domains. F* is also verification-oriented as it applies formal verification techniques to prove the correctness of programs written in it. Finally, F* is effectful, for it allows programs with side-effects, such as I/O, exceptions, stateful processing, or programs that potentially diverge.

### 2.5.1 Introduction to F*: Syntax, Type and Effect

As a functional programming language, F* shares most concepts with Haskell. For example, F* is also strongly typed and pure. Syntactically, there are only small differences between F*'s syntax and that of Dependent Haskell. Recall the `double` function

introduced in section 2.1 that doubles every element in the list. Below is an example of F*'s syntax:

```
val double: list int -> Tot (list int)
let rec double l = ...
```

In F*, programmers also specify the function signature before the actual implementation. All function signatures start with the keyword `val` and the type signature is introduced after a single colon instead of a double colon in Haskell. Also, all function implementations start with the keyword `let` and if the function is recursively called, the keyword `rec` will be specified between `let` and the function name. In the `double` example above, the function takes in a list of integers and returns another list of integers where each element is doubled.

Also it worth mentioning that the return type in the function signature is not just the return type `list int` but a effect-and-type pair `Tot (list int)`. The prefix `Tot` is called an effect. Any function with a return type of `Tot typ` is unconditionally pure, guaranteed to evaluate to a term of type `typ` without any possibility to enter an infinite loop. This `Tot` effect specifies the function `double` to be total, or unconditionally pure. It is a special derived form of the `Pure` effect, categorizing all pure functions that always terminate without modifying any program state. As an effectful language, F* categorizes programming effects and in addition divergence in its type system. Here we briefly introduce the concept of an effect, which will be discussed in more detail in section 2.5.4. For example, the effect `Pure` on total functions is F*'s logical core [17]. The soundness (or correctness) of every pure function is determined by the required termination checks performed by the F* compiler. In addition, impure functions will be annotated with the corresponding effect. For example, `Dv` effect, a derived form of the general `Div` effect, is used to categorize all potentially nonterminating functions [17].

### 2.5.2 Simple Inductive Types

Similar to Dependent Haskell, F* allows creating new types using the `type` keyword. We show the definition of the standard library function `list` as follows:

```
type list (a:Type): Type =
  | Nil  : list a
  | Cons : a -> list a -> list a
```

The list definition in F* is similar to a `vec` GADT definition in Dependent Haskell. There are several new syntax that worth mentioning in the above `list` definition in F*. First, datatype in F* is defined in all lower case. Second, one applies the type parameter `(a:Type)` before colon instead of as the first type parameter after the colon. The type `a` is still considered the first type parameter of the return type, as can be seen in the constructed type `list a` of the empty list constructor `Nil`. The reason to declare `(a:Type)` before colon is to bring the generic type `a` in scope for all type definitions below. Third, notice that the equality sign `=` is used instead of `where` to start the actual type definition and also notice that both type constructors begin with a capital letter. The

definition of each constructor is preceded by the `|` keyword and the constructor types follow after a single colon. Finally, as we introduced before, F* requires all return types to annotate effects, but we didn't specify any effect for the two constructors `Nil` and `Cons`. When effects aren't specified, the default effect `Tot` is automatically inferred by F* for the result `list a`.

With the `list` definition, the `length` function can be easily programmed to return the length of any input list. Specifically, one applies pattern matching on the input list `l` with syntax `match l with` and uses the `|` symbol to separate cases. The `length` function defined on `list` is widely used in the refinement logic to specify conditions for proving lemmas.

```
val length: #a:Type -> list a -> Tot nat
let rec length l = match l with
  | [] -> 0
  | _ :: tl -> 1 + length tl
```

There are two new syntax introduced in the `length` definition, two syntactic sugar for the list constructors where `[]` represents `Nil` and `_ :: tl` represents `Cons _ tl`. In the first pattern matching case, one defines the length of an empty list, `Nil`, to be `0`. In the second pattern matching case, one recursively defines the length of a list to be one more than the length of the sublist starting from the second element. The `_` symbol is commonly known as a wildcard, representing the value we don't care about when performing pattern matching.

### 2.5.3 Intrinsic and Extrinsic Style of Verification

F* in general supports two approach for verifying programs. Programmers can prove properties either by enriching the types of a function or by writing a separate lemma. These two ways towards verification are often called the intrinsic and extrinsic style.

#### 2.5.3.1 Proving List Concatenation Intrinsically through Type Enrichment

Consider again the `concat` example as introduced in section 2.4.5. To concatenate two lists, one wants to prove the property that the length of the resulting list is equal to the sum of the lengths of the two input list. Here is the definition of `concat` that concatenate two lists:

```
let rec concat l1 l2 = match l1 with
  | Nil      -> l2
  | hd :: tl -> hd :: (concat tl l2)
```

***Type Enrichment through Dependent Types***

Similar to what we have discussed in detail on Dependent Haskell's verification approach, F* supports verifying the property by indexing on the length of both input list and output list using dependent types. First we introduce that definition of the `vec` GADT.

```
type vec (a:Type): nat -> Type =
  | Nil  : vec a 0
  | Cons : a -> nat -> vec a n -> vec a (n + 1)
```

This definition of the dependently typed `vec` is analogous to the `Vec` GADT in Dependent Haskell as defined in section 2.4.2. Again, this GADT is dependently typed because it depends on the value of its parameter, the length of a list of type `nat`. The type `a` of kind `Type` represents the type of each element in the list. In the actual definition, the type `a` can represent `int` or `bool` as long as all the `a`s match up. The `nat` type, again represents the length of the list. Finally, the constructed type is a GADT of kind `Type`. In regards to the type constructors, `Nil` constructs an empty list of length `0` and `Cons` adds one more element to the existing list to obtain a new vector of length one more than the previous list.

There is a variation to the Dependent Haskell implementation in section 2.4.2 that worth discussion. Notice that we directly use the type `nat` without defining a new GADT for it. Also, as a result, instead of using the `Zero` and `Succ` type constructors as in Dependent Haskell, intuitive natural numbers `0`, `n` and `n+1` can be directly applied in the type. The convenience is a direct result of F*'s support on refinement types. Specifically, the type of natural numbers is defined by refining the integer type with an additional predicate. The predicate ensures that the integers are all greater than or equal to `0`, so the natural numbers are defined as `x:int{x>=0}`. Since the natural number type is very common in application, F* also defines an abbreviation `nat` to represent the refinement type. We say that `nat` is a subtype of `int` because every term of type `nat` also automatically has type `int`.

Finally, using the defined `vec` datatype, the verified `concat` function is very similar to Dependent Haskell.

```
val concat : #a:Type -> #n:nat -> #m:nat -> vec a n -> vec a m -> Tot (vec a (n
    + m))
```

We want to verify that concatenating two lists, each of length `n` and `m`, will result in a list of length `n + m`. The dependent type encoding is the same as that in Dependent Haskell. There are, though, two syntactic differences in the F* version. First, notice that in the above implementation, three extra arguments are introduced in the type definition, all are preceded by the pound sign `#`. The first argument `#a:Type` brings the generic type `a` in scope to the `concat` function. The second and third argument `#n:nat` and `#m:nat` is responsible to bring the two input lengths in scope. When preceded by the `#` symbol, the types are turned into what's known as implicit arguments. Implicit arguments tell the F* type system to try inferring the type automatically instead of always requiring programmers to provide them manually in the recursive calls. It is a kind of type inference that ease the complexity of writing code in a strictly typed environment.

Second, notice that instead of defining and using type family as in Dependent Haskell, we can directly apply the plus operator to specify the length of the result list. Addition, represented as the symbol `+`, is a total function defined in the library that is guaranteed to terminate. Hence F*'s type system can directly bring `+` in logic and use it in the `vec`

GADT.

***Type Enrichment through Refinement Types***

Recall that F* supports both dependent types and refinement types. Therefore, going back to the weak typing using `list`, the same property can be verified by giving `concat` the following type:

```
val concat: #a:Type -> l1:list a -> l2:list a -> Tot (l:list a{length l =
    length l1 + length l2})
```

Notice how concise it is to enrich the function type using refinement types. All it is required additionally is the intuitive formula of the property of interest, `len l = len l1 + len l2`.

#### 2.5.3.2 Proving List Concatenation Extrinsically through Lemmas

In addition to the intrinsic style of verification, F* also supports extrinsic verification through lemmas. Consider the following type signature giving to the `concat` function,

```
val concat: #a:Type -> l1:list a -> l2:list a -> Tot (list a)
```

With the predefined `length` function, one can prove the same property as in section 2.5.3.1 using a lemma as shown below:

```
val concat_pf: #a:Type -> l1:list a -> l2:list a
            -> Lemma (requires True)
                     (ensures (length (concat l1 l2) = length l1 + length l2)))
let rec concat_pf l1 l2 = match l1 with
    | []     -> ()
    | hd::tl -> concat_pf tl l2
```

Similar to the plus operator introduced in the last section, `concat` is a total function that proved to terminate. Hence, it can be directly lifted wholesale to the logic level and be reasoned about in the postcondition following the keyword `ensures`. In the `concat_pf` lemma, the property is valid in all conditions, so the precondition following the keyword `requires` is simply defined as `True`. With both precondition and postcondition specified, one can prove the lemma recursively. The `()` symbol introduced here is equivalent to `Refl` in Dependent Haskell, short for reflexivity, and denotes the satisfaction of the property. Finally, the proof on nonempty list follows directly from inductive hypothesis.

### 2.5.4 Proof Automation

Various research has been done on proof automation to reduce the burden of manual proofs. Automated theorem proving is a subfield of mathematical logic that proves mathematical theorems through computer programs. Satisfiability Modulo Theories (SMT) solvers provide a good foundation for highly automated programs [14].

Satisfiability Modulo Theories (SMT) problems are a set of decision problems that generalize boolean satisfiability (SAT) with arithmetic, fixed-sized bit-vectors, arrays, and other related first-order theories. Z3 is an efficient SMT Solver introduced by Microsoft Research that is now widely adopted in program verifications [6]. Dependent Haskell doesn't support proof automation at any level, but the F* language achieves semi-automation through a combination of SMT solving using Z3 and user-provided lemmas.

### 2.5.5 Termination Checking

As we are mostly interested in how F* handles potentially divergent functions, we will focus on the `Tot` and `Dv` effect and introduce F*'s approach towards termination checking.

F* by default requires termination checking for every function, since divergent functions may generate unsound verification conditions (VCs) and eventually end up proving some erroneous theorems. F* introduces a new termination criterion based on a well-founded partial order $\prec$ over all terms. A partial order is a binary relation on a set that is reflexive, antisymmetric and transitive. For example, the `<=` ordering on integers is a partial order, since for every integer `a`, `b` and `c`, it follows that `a <= a` (reflexive), `a <= b` but `b >= a` (antisymmetric), and finally if `a <= b` and `b <= c` then `a <= c` (transitive). F* insists each iteration of a recursive function be applied on a strictly smaller domain, such that the parameter of the function is guaranteed to decrease in the defined partial order during each execution. For example, consider the total function `length` introduced in section 2.5.2. In the second definition where we pattern match on a nonempty list, i.e. `_ :: tl -> 1 + length tl`, the function is recursively called on the tail of the input list `tl`. Since the tail of the list has one less element than the original list `l = _ :: tl`, F* compiler completes termination checking and accepts this definition.

On a high level, the F* type-checker provides four built-in well-founded ordering that can be automatically applied in proving termination, but it also accepts decreasing metrics explicitly provided by the programmer following the `decreases` keyword.

Once the function is proved total, one can specify it using the `Tot` effect, notifying the F* effect system that the function is guaranteed to terminate. If, instead, one can't find any termination metric or the function diverges, F* provides the `Dv` effect, i.e. divergent, to categorize all these potentially divergent functions. Suppose that we define our `length` function as follows:

```
let rec double l = double l
```

This new definition is obviously not going to terminate. In fact, it will keep calling itself and loop forever. In this case, if we want to compile the program for some reason, we have to mark the function return type with the `Dv` effect, as in `val double: list int -> Dv (list int)`. Once a function is marked as `Dv`, F* will turn off all termination checking on that function. In general, since the soundness of F*'s type system depends on the termination checking, `Dv` effect should be used sparingly. The above example is just to illustrate the idea of `Dv` effect, but this effect is only used for functions that actually

diverges or requires too much effort, if not impossible, to prove termination.

# 3 Approach and Novelty

To gain a better understanding of the language design decisions and verification techniques in Dependent Haskell and F*, we first studied the previous literature on each language. In general, our assessments of Dependent Haskell and F* are based on implementing and proving a wide spectrum of algorithms in both languages. These programs are written so that we can have a deeper experience with both languages and explore their features more thoroughly.

We start from implementing the datatype of a length-indexed vector in both Dependent Haskell and F*. Then we verify various functions that support types of finite length vectors from Haskell's `Data.List` module, including `length`, `head`, `tail`, `init`, `last`, `snoc`, `reverse`, `and`, `or`, `null`, etc. We then focus on the verification for various sorting algorithms, such as insertion sort, mergesort and quicksort. Finally, we examine the capability of proving divergent functions through three examples. The first example is a proof of the Peano division property `(m * n) / n = n` for the peano naturals `m` and `n` using the divergent definition of division with zero divisors. The second proof is the equivalence of multi-step and big-step evaluators in simply typed lambda calculus. Finally, we try to verify some obviously invalid proofs in each language and investigate how non-termination might interfere with normal type checking process at compile time.

We expect to structure the comparison mainly focusing on two aspects: user interface and approaches towards potentially non-terminating functions. As far as we know, we are the first researchers to directly compare the two languages Dependent Haskell and F*.

User interface refers to the interaction between programmers and programs, specifically programming simplicity when implementing programs. We are interested in examining how these two language designs differ in respect to the ease of use measured by syntactic verbosity and library/documentation resources. As both languages are verification oriented, in addition to general programming syntax, we will also include a comparison on proof complexities for each verification technique. We believe that a good programming language should provide a clear, concise and user-friendly interface with a relatively detailed library and documentation support.

These two approaches towards potentially non-terminating functions is measured by the strength and completeness of the type systems. A type system is said to be reliable if every property that passes compile time type check is logically valid. That is to say that every verification is valid as long as the program compiles. Conversely, a type system is said to be complete if every logically valid property with respect to the system's semantics can be encoded and verified in the system. Both type systems in Dependent Haskell and F* are reliable and complete when we restrict the range of functions to those that are provably terminating, called total functions. However, when we extend the system to consider potentially non-terminating functions, either functions that diverge or are provably hard to prove convergent, ensuring both strength and completeness in the type

system is a huge challenge. Without compile time termination checking, Dependent Haskell seems to be complete and is able to prove properties on logically valid properties for divergent functions. Yet its type system is not reliable enough, as it type checks some verifications on logically erroneous statements and further requires run time checking to ensure program correctness. On the other hand, F* is provably reliable, since one can ensure the termination of all total functions of the Tot effect and the correctness of verification follows. However it is unknown whether F*'s type system is complete with its categorization of potential divergent functions using the `Dv` effect.

# 4 Current Results

As summarized in Section 3, we have implemented various functions that support types of finite length vectors from Haskell's `Data.List` module, three machine-checked sorting algorithms, two proofs on properties from potentially divergent functions, and finally some erroneous proofs to test the strength of each type system.

## 4.1 User Interface: General Language Design and Support

In terms of user interface, we mainly focus on comparing three aspects of Dependent Haskell and F*. In Section 4.1.1, we discuss the general language design and compare the syntactic verbosity in each language. Section 4.1.2 focuses on library and documentation resources. Finally, we present the Insertion Sort algorithm as an example to analyze the verification techniques supported by each language.

### 4.1.1 Syntactic Verbosity

Throughout our analysis, we conclude that Dependent Haskell is syntactically verbose while F* has a fairly friendly user interface.

#### 4.1.1.1 Dependent Haskell

Dependent Haskell enforces phase separation between the term and type level. Thus, to apply a function defined in the term level to a dependent type in the type level, programmers are often expected to repeat implementations of the same logic.

The first type of duplication is in data type definitions. Dependent Haskell enables dependently typed programming through the use of singletons. Singleton is a special type, also known as a singleton type, that bridges the gap between run-time values and compile-time types. Each singleton type is indexed by a type variable of the promoted kind (see section 2.4.2 on definition of kind). For example, `SNat :: Nat -> Type`, a GADT indexed by a type of kind `Nat` and returns a new type `Type`(or `*`), is a singleton type for the `Nat` datatype on natural numbers. A singleton type is a type with exactly one value, where each type variable indexing the singleton type is uniquely mapped to the term of that type. Hence there is an isomorphism between one singleton type and its associated values, ensuring that term-level computation and type-level computation always go hand in hand [8].

Take the natural numbers as an example, there are three different definitions – the original datatype `Nat`, the kind `'Nat`, and the derived singleton type `SNat`. In 2012, Yorgey et al. introduced the datatype promotion mechanism that automatically promotes datatypes up a level to kinds and the data constructors to type-level data [21]. Hence, the kind definition `'Nat` can be automatically derived from the original datatype, reducing programming redundancy down by a level. However, prior to the introduction of the `singletons` library, programmers still need to manually implement the singleton type, even though the definition of a singleton type is a straightforward extension to that of the original datatype.

```
-- original datatype Nat definition
data Nat :: * where
  Zero :: Nat
  Succ :: Nat -> Nat

-- singleton type SNat definition
data SNat :: Nat -> * where
  SZero :: SNat Zero
  SSucc :: SNat n -> SNat (Succ n)
```

As shown above, `Zero` and `Succ` are both automatically promoted type-level data from the `Zero` and `Succ` data constructors. Also, the `SZero` and `SSucc` data constructors are indexed on the promoted `Nat` type and help specify the dependency between the compile-time type parameters (`Zero`, `Succ Zero` etc) and the run-time term values (`Zero`, `Succ Zero` etc).

The `singletons` library introduced by Eisenberg and Weirich aims to remove the above code duplication. Using Template Haskell, the library can automatically generate the definitions for singleton types. However, the programming overhead still remains, as the library only frees programmers from the tedious singleton definitions but doesn't hide the nature of the internal singletons encoding. Programmers are still expected to have a deep understanding of the phase separation and the singleton types, and to take full charges of the function implementations using the library-generated definitions.

The second type of duplication comes from function definitions. Functions such as adding two natural numbers, need to be specified once in the term level for the original datatypes, once in the type level using type families, and once in the term level for the singleton types.

```
-- term-level function plus definition
plus :: Nat -> Nat -> Nat
plus Zero    n  = n
plus (Succ m) n = Succ (plus m n)

-- type-level type family Plus definition
type family Plus (m :: Nat) (n :: Nat) :: Nat where
  Plus Zero    n   = n
  Plus (Succ m') n = Succ (Plus m' n)

-- term-level function sPlus definition for singleton types
sPlus :: SNat m -> SNat n -> SNat (Plus m n)
sPlus SZero    n   = n
sPlus (SSucc m) n = SSucc (sPlus m n)
```

As shown above, we define the term-level `plus` function, adding two natural numbers and obtaining the result of their sum. The base case is defined, in mathematical terms, as `0 + n = n` and the inductive case as `(1 + m) + n = 1 + (m + n)`. Similarly, following the exact same logic, we define the type family `Plus` for adding two natural numbers in the type level. That is to say that instead of using the `plus` function in the actual implementation, the `Plus` type family can be applied to the type signature as a quantifier over the dependent types. Finally, we implement the function `sPlus` on the singleton types. The only difference between the definition of `sPlus` and that of `plus` is that the operation is quantified over the type parameter of the singleton type instead of directly performing on the terms. Specifically, pattern matching on the singleton type `SNat m` gives us insights into both the type variable of kind `Nat` and the associated term-level variable of type `Nat`.

The above three versions of the same addition algorithm on natural numbers add in lots of additional burdens when using the language. To free programmers from these repetitive and tedious works, the `singletons` library also automates the function definition process. Through Template Haskell, the library supports promoting term-level functions and directly reuse them in the type level. It also supports enriching functions with richer types, extending the term-level definitions to directly apply on singleton types.

With the `singletons` library, now we only need to provide the original data type definition for `Nat` and the term-level function definition for `plus` as shown below:

```
$(singletons [d|
  data Nat = Zero | Succ Nat
    deriving Eq

  plus :: Nat -> Nat -> Nat
  plus Zero    n = n
  plus (Succ m) n = Succ (plus m n)
  |])
```

The above code follows the Template Haskell syntax. Template Haskell is an extension to Haskell that supports compile-time meta-programming. The code snippet starts with the `$` symbol and follows by the normal Haskell datatype and function definitions

enclosed in the quoting syntax `[d| ... |]`. This `$()` syntax is known as a Template Haskell splice, whose contents are evaluated at compile time [8]. Specifically, in the natural numbers addition example, the `singletons` function takes in the Template Haskell quote containing an abstraction of both the datatype definition on `Nat` and the function definition on `plus`. It then generates a list of Template Haskell declarations including the three definitions – `SNat`, `Plus`, `sPlus` – introduced before. Finally, these Template Haskell declarations are inserted back into Dependent Haskell and are defined to use in all program verifications.

The `singletons` library reduces programmer's burden to manually implement all functions at different level through the automation of singleton data type definition, and the generation of type family and enriched functions on singleton types. Yet, the library still doesn't resolve the syntactic verbosity in Dependent Haskell. The `singletons` library only helps lessen the effort users need to spend on the duplicated declarations, but it doesn't abstract away any detail of the actual definitions. In addition, the generated functions contain a lot of renaming for keywords, such as `case`, terms and types. As a result, all the generated data types and functions after turning on the `ddump_splices` compiler option are very hard to read. In turn, compare to debugging using self-defined data types and functions, proofs using the generated structures sometimes add additional complexity to parse error messages. However, programmers still need to fully understand how type family and singletons work to compile their programs and to complete the verifications. Therefore, even with the assistance of both the datatype promotion technique and the `singletons` library, Dependent Haskell's syntax remains a little clumsy and is not very friendly to users new to programming verification.

#### 4.1.1.2 F*

On the other hand, F* is syntactically concise. It supports both type-level specifications through dependent types and type-level computations through refinement types. For example, the type signature for the `concat` function introduced in section 2.4.5 can be specified using dependent type as `val concat: #a:Type -> #n:nat -> #m:nat -> Vec n a -> Vec m a -> Vec (n + m) a`. F*'s approach towards dependent types is similar to that of Dependent Haskell, except that F* frees programmers from the repetitive type family implementation of `Plus`. Through the `T-Ret` typing rule using the idea from monadic returns (we won't go into detail for the typing rule) [17], F* can automatically enrich the types of a total function and reflect the existing term-level implementations into logic for reasoning. Specifically in the example of `concat`, the library function `(+)` is automatically reflected to the type level and is applied in the dependent type `Vec (n + m) a`.

The same verification for concatenating two lists can also be specified using refinement type, as `val concat: #a:Type -> #n:nat -> #m:nat -> l1:Vec a -> l2:Vec a -> r:Vec a{len r = len l1 + len l2}`. Instead of indexing the length of the vector to form dependent types, we can directly specify the condition `len r = len l1 + len l2` as a logical predicate to the resulting type.

#### 4.1.1.3 Summary

Question: would a summary here be helpful for readers? Since the key point of this section is very clear and is already mentioned in the very beginning, would a summary be necessary still? Especially given that this subsection is relatively long...

### 4.1.2 Library and Documentation Supports

Dependent Haskell has many well-documented and timely-maintained libraries as well as a lot of online resources. However, the relatively new F* language is still in the process of building up libraries and adding documentation.

Dependent Haskell, extending Haskell with dependent types, has been officially implemented in GHC 8.0 and is now a part of Haskell. As Haskell is a mature, industrial-strength programming language, Dependent Haskell inherits all its resources. Specifically, Haskell provides a central package archive called Hackage where packages are introduced, maintained, and supported. In addition, Hoogle is a holistic search engine for Haskell APIs which provides advanced queries through function names such as `sum`, and even through approximate type signatures such as `int -> int -> int`. As Haskell becomes more popular in the real world, there emerges a plethora of online tutorials and textbooks with detailed explanations of syntax, concepts and algorithms. Since the introduction of Dependent Haskell in 2014, tutorials, such as schoolofhaskell, have gradually incorporated Dependent Haskell into their documentations. Official resources for Dependent Haskell are also available on Richard Eisenberg's Github repository for his PHD dissertation (https://github.com/goldfirere/thesis).

However, as a completely redesigned language that is relatively new, F* does not have many resources by now. Besides various conference publications, F* has an official website organizing all of its resources (https://www.fstar-lang.org). Most of the documentations are through its interactive tutorial (https://www.fstar-lang.org/tutorial/) and examples in the official FStar Github repository (https://github.com/FStarLang/FStar). In addition, source code for each library function can be found in the `ulib` folder of its Github repository (https://github.com/FStarLang/FStar/tree/master/ulib). There are a lot of functions implemented, but they are mostly shown as the original implementation without much additional documentation.

### 4.1.3 Proof Complexity

Regarding verification techniques, F* provides a very intuitive user interface that engages users in the process of writing proofs. In addition, the SMT solver provides great automation that frees users from many non-interesting proofs. On the other hand, theorem proving in Dependent Haskell has a steep learning curve and is already laborious for some basic sorting algorithms.

The current Dependent Haskell as implemented in GHC 8.0 doesn't support any proof interaction or automation. Verification in Dependent Haskell is expressive, but could get really tedious even for relatively simple sorting algorithms, like insertion sort, mergesort or quicksort. F*, on the other hand, provides a flexible combination of SMT-based

proof automation and manually constructed proofs. In addition, F* can use SMT solver, such as Z3, to match SMT patterns provided in the manually provided lemmas to assist with user-constructed proofs. F* supports theorem proving through either enriching function types (officially referred to as the intrinsic style) or by writing separate lemmas on properties (officially referred to as the extrinsic style) [17]. It categorizes effects into `Tot`, `Dv`, etc, requires explicit annotation of the effect in type signatures and enforces termination checking on all total functions.

In this section, we will focus on comparing how Dependent Haskell and F* vary in their approaches to prove properties on insertion sort. It is a simple sorting algorithm that builds the result sorted list by inserting each input element to the correct position one at a time. The insertion sort algorithm is chosen in this comparison for the following three reasons. First, it is a widely-used algorithm that is easy to understand for most computer science majors. Second, the correctness of the sorting algorithm is well-defined and important to be verified. Third, it features an algorithm with nontrivial properties that are hard to prove intrinsically. Specifically, to prove that a sorting algorithm is correctly implemented, we need to check both contents and ordering of the result list. The contents should remain the same as the input list, with no additional or missing elements. It is equivalently to say that the output list is a permutation of the input list. Also the result list must be verified to be sorted, meaning that it has a monotonically non-decreasing order.

To ensure the fairness of the comparison, verification in both languages is based on a same insertion sort definition. Also without loss of generality, we abstract out the details of generic types and focus on discussing verification on a list of natural numbers in this paper. The base algorithm for insertionSort is shown below and we will discuss it in detail when talking about the proofs.

```
insert :: Nat -> [Nat] -> [Nat]
insert n []    = [n]
insert n (h:t) = if n <= h then (n:h:t) else h:(insert n t)

insertionSort :: [Nat] -> [Nat]
insertionSort []    = []
insertionSort (h:t) = insert h (insertionSort t)
```

In the following sections, we present the proof in both languages and will discuss the comparison at the end. Section 4.1.3.1 features the implementation in Dependent Haskell and section 4.1.3.2 features that in F*.

#### 4.1.3.1 Dependent Haskell

Our verification of insertion sort in Dependent Haskell defines two GADT `NonDecProof` and `PermutationProof`, each encoding one property that we want to show. The actual proof, as shown below takes in a singleton list and returns a tuple of proofs. By using a singleton list `SList lst`, Dependent Haskell extracts the term level list `lst` to the type level and allows the list to be reasoned in the type signature.

```
insertionSort_pf :: SList lst -> (NonDecProof (InsertionSort lst),
                                  PermutationProof lst (InsertionSort lst))
insertionSort_pf slst = (nondec_pf slst,
                         permutation_pf slst)
```

### *Proof of Non-decreasing Order*

First, we discuss the proof of non-decreasing order, i.e. the lemma `nondec_pf` that takes in a singleton list and returns the defined `NonDecProof` that ensures the result list after insertion sort has a non-decreasing ordering.

Below we introduce the definition of the `NonDecProof` GADT:

```
data LeqProof :: Nat -> Nat -> Type where
  -- a proof of 0 <= a
  LeqZero :: LeqProof Zero a
  -- a proof that if a <= b then a + 1 <= b + 1
  LeqSucc :: LeqProof a b -> LeqProof (Succ a) (Succ b)

-- A proof term asserting that a list of naturals is in non-decreasing order
data NonDecProof :: [Nat] -> Type where
  NonDecEmpty :: NonDecProof '[]
  NonDecOne   :: NonDecProof '[n]
  NonDecCons :: LeqProof a b -> NonDecProof (b ': rest) -> NonDecProof (a ': b
      ': rest)
```

The proof contains three value constructors `NonDecEmpty`, `NonDecOne` and `NonDecCons`. The first two are trivial cases, stating that the empty list and a single-element list is sorted (or has a non-decreasing ordering). Consider the value constructor `NonDecCons`, we first take in another GADT `LeqProof a b` ensuring the property `a <= b`, then recursively verify if the list starting with element `b` is non-decreasing. Given that we have a proof of `a <= b` and a proof that the list `b:rest` (i.e. `[b,c,d,e,...]` where `rest = [c,d,e,...]`) is non-decreasing, we can construct a proof term ensuring that the list `a:b:rest` is also non-decreasing.

Given the self-defined proof term `NonDecProof`, we proceed to implement the lemma `nondec_pf` as follows:

```
-- A lemma that ensures that the result of an insertion sort has a
    non-decreasing order
nondec_pf :: SList lst -> NonDecProof (InsertionSort lst)
nondec_pf lst = case lst of
  SNil -> NonDecEmpty
  SCons h t -> case nondec_pf t of
    nondecPf_t -> insert_nondec_pf h (sInsertionSort t) nondecPf_t
```

The lemma takes in a singleton list and returns the `NonDecProof`, ensuring that the list after an insertion sort has a non-decreasing order. Recall from section 4.1.1.1 that a singleton type is a type with exactly one value, where each type variable indexing the singleton type is uniquely mapped to the term of that type. By using a singleton list of

type `SList lst` instead of a normal list of type `List`, we can ensure that the list `lst` that we are proving on in `NonDecProof` is exactly the same list as the input list. As such, we can extract out the `lst` term from the input and reason about it in the type level.

Looking at the actual definition of `nondec_pf`, we pattern match on the input list and separately discuss two cases – the empty list `SNil` and the nonempty list `SCons h t`. If the input list is empty, then it is immediate that insertion sort on an empty list returns an empty list that by default has a non-decreasing order. Otherwise, when the input list is nonempty, we proceed to check if insertion sort on the tail of the list `t` produces a non-decreasing list. If so, we apply the additional lemma that ensures the fact that inserting into a non-decreasing list produces another nondecreasing list.

Finally, we encode the helper lemma `insert_nondec_pf` as follows:

```
-- A lemma that states that inserting into a non-decreasing list produces
    another non-decreasing list
insert_nondec_pf :: forall n lst. SNat n -> SList lst -> NonDecProof lst ->
    NonDecProof (Insert n lst)
```

This lemma states that inserting any element into any non-decreasing list produces another non-decreasing list. The `forall n lst.` quantifier is the universal quantifier that ensures that the property works for any element and any provably non-decreasing list.

Below we discuss the actual implementation of the `insert_nondec_pf` lemma step by step.

```
insert_nondec_pf n lst nondecPf_lst = case nondecPf_lst of
  NonDecEmpty -> ...
  NonDecOne -> ...
  NonDecCons hLeqH2 nondecPf_t -> ...
```

On a very high level, this lemma pattern matches on the non-decreasing proof of the input list and separately prove on each of the three terms. The value `NonDecEmpty` represents a non-decreasing empty list; the value `NonDecOne` represents a non-decreasing single-element list; finally, the value `NonDecCons hLeqH2 nondecPf_t` represents a non-decreasing list of at least two elements.

### *Case 1: NonDecEmpty*

```
NonDecEmpty -> NonDecOne
```

If the input list is empty then inserting an element results in a list of single element, which is a non-decreasing list by the definition of `NonDecOne`.

### *Case 2: NonDecOne*

```
NonDecOne -> case lst of
  SCons h SNil -> case sLeq n h of
    STrue -> NonDecCons (sLeq_true__le n h) NonDecOne
    SFalse -> NonDecCons (sLeq_false__nle n h) NonDecOne
```

If the input list has one element, then we proceed to check if `n <= h`, i.e. if the inserting element is less than the head of the list. In both cases, we are done by applying the `NonDecCons` constructor with the associated `LeqProof` proof. There is one more syntax that worth mentioning. In the implementation above, we use the two functions `sLeq_true__le` and `sLeq_false__nle` to generate the associated proof terms of type `LeqProof` that are passed to the `NonDecCons` constructor.

***Case 3: NonDecCons***

```
NonDecCons hLeqH2 nondecPf_t -> case lst of
  SCons h t -> case sLeq n h of
    STrue -> NonDecCons (sLeq_true__le n h) nondecPf_lst
    SFalse -> case sLeq_false__nle n h of
      hLeqN -> case insert_nondec_pf n t nondecPf_t of
        nondecPf_nt -> case t of
          SCons h2 _ -> case sLeq n h2 of
            STrue -> NonDecCons hLeqN nondecPf_nt -- since we have h <= n and
                t(h2:_) is non-decreasing
            SFalse -> NonDecCons hLeqH2 nondecPf_nt -- since we have h <= h2
                and t(h2:_) with n inserted is non-decreasing
```

The last case when the input list has at least two elements is more complicated. We first obtain the head and tail of the input list by pattern matching on the input list. Then, similar to the second case, we check if `n <= h` and we are done if so. Otherwise, we recursively check if adding the element `n` to the tail of the list `t`. If so, we obtain a proof that inserting `n` to the tail of the list results in a non-decreasing list. We proceed to obtain the head `h2` of the sublist, i.e. tail `h` through pattern matching and checks if the element we want to insert is less than or equal to `h2`. If so, we can conclude that the resulting list `h:n:h2` is non-decreasing by the definition of `NonDecCons`. If not, then we can also conclude that the resulting list `h:h2` is non-decreasing by `NonDecCons`, since `h <= h2` and the list after inserting `n` into `h2` is non-decreasing.

***Proof of Permutation***

Next, we discuss the second proof that the list after insertion sort is a permutation of the input list, i.e. the lemma `permutation_pf` that takes in a singleton list and returns the defined `PermutationProof`.

Below we introduce the definition of the `PermutationProof` GADT:

```
-- A proof term asserting that l1 and l2 are permutations of each other
data PermutationProof (l1 :: [k]) (l2 :: [k]) where
  PermId  :: PermutationProof l l
  PermIns :: InsertionProof x l2 l2' -> PermutationProof l1 l2 ->
             PermutationProof (x ': l1) l2'
```

The `PermutationProof` encodes a proof that ensures the original list `l1` and the list `l2` after insertion sort are permutations. We say that two lists are permutations of each other if they contain the same set of elements. The proof contains two cases. The trivial case `PermId` states that the same two lists are permutations. The inductive case `PermIns`

states that given that `l1` is a permutation of `l2`, if we can insert an element `x` to `l2` and get a new list `l2'`, then the result list after inserting `x` to `l1` is a permutation of `l2'`. In short, adding the same element to both lists that originally contain the same set of elements will result in two lists that remain a permutation of one another. InsertionProof is a proof term asserting that `l2'` is the list produced when `x` is inserted anywhere into the list `l2`.

Next, given the proof term `PermutationProof` defined above, we can implement the lemma `permutation_pf` as follows:

```
-- A lemma that states that the result of an insertion sort is a permutation
-- of its input
permutation_pf :: SList lst -> PermutationProof lst (InsertionSort lst)
permutation_pf lst = case lst of
  SNil -> PermId
  SCons h t ->
    case insert_permutation_pf h (sInsertionSort t) of
      insertionPf -> PermIns insertionPf (permutation_pf t)
```

Lemma `permutation_pf` takes in a singleton list and returns the proof ensuring that the result of an insertion sort is a permutation of its input. Similar to all lemmas introduced before, this lemma also starts with pattern matching on the input list. If the input list is empty, then we can immediately conclude that two empty lists are permutations by `PermId`. Otherwise, when we have a nonempty input list of structure `h:t`, we first verify through another lemma that inserting `h` into the insertion sort of `t` returns a correct new list `l'` with `h` inserted. Then, we recursively check if the tail `t` is a permutation of the insertionsort of `t`. Finally, we can directly construct the proof term using the `PermIns` value constructor as defined. To check if inserting an element works correctly, we implement the lemma `insert_permutation_pf :: SNat n -> SList lst -> InsertionProof n lst (Insert n lst)`. The lemma makes sure that inserting an element `n` into a list `lst` produces the same list as the result of calling `Insert n lst`.

In the previous two sections, we have proved that the result list is both sorted and a permutation of the input list. As a result, since the above Dependent Haskell encoding of insertion sort compiles, we can conclude that insertion sort is verified to be correct, meaning that it meets all specification in full.

Question: would adding a summary in diagram/chart be helpful here to recap? I am worried about my readers getting overwhelmed by the details but lose the whole picture.

#### 4.1.3.2 F*

Since F* supports refinement types, we don't have to define our own proof terms as in the proof in Dependent Haskell (though proving using dependent types is also supported). Instead, we can directly prove the property intrinsically by providing refinement conditions for `insert` and `insertionSort`.

Let's first take a look at the function signature of function `insertion_sort`.

```
val insertion_sort : l1:list nat -> Tot (l2:list nat{sorted l2 /\ (forall k.
    elem k l2 == elem k l1)})
```

The above `insertion_sort` function takes in a list of natural numbers and returns another list of natural numbers. Recall that there are two properties that we want to show in an insertion sort, i.e. the contents and the ordering of the result list. Using refinement types, we specify that the result list is sorted through `sort l2` and we prove the permutation lemma by stating that `forall k. elem k l2 == elem k l1`. There are a lot of new syntax introduced here, so let's discuss one by one (First time readers may skip bullet point 3 and 4):

1. The `sorted` function is a helper function that prove the first sorted lemma on our result list `l2`. We will discuss the proof later in detail.

2. The `forall k.` quantifier is the universal quantifier. So our permutation lemma states that every element in the result list `l2` is also an element in the input list `l2`.

3. The usage of the `==` symbol is different from its usage in Java or C/C++. Instead of connecting two booleans, the `==` symbol is the type constructor for an equality predicate. It helps to compare the equality between two types and has signature `val (==) : #a:Type -> a -> a -> Tot Type0` . On the other hand, F* compares a pair of booleans using `=`.

4. Finally, F* also provides the propositional connective $\wedge$ symbol for conjunction, which connects two types similarly to how `&&` connects two booleans.

The last two part of our discussion is not expected to be understood by everyone at a first read. The key point here is that the predicate provides in the refinement type is in fact a type, not a boolean term. But for now, you may think of the symbols `==` just as how it works to compare booleans and the symbol $\wedge$ just as the logical `&&`.

Next we define the function `insertion_sort` in F*, as what we've shown in the beginning of section 4.1.3.

```
let rec insertion_sort l1 = match l1 with
    | []       -> []
    | hd :: tl -> insert hd (insertion_sort tl)
```

In order to understand why and how the refinement type provided type checks (how it carries out the proof), we need to introduce the helper lemmas `sorted` and `elem`. The type signatures of both is provided as follows:

```
val elem: nat -> list nat -> Tot bool
val sorted: list nat -> Tot bool
```

Function `elem` takes in a natural number element and an input list of natural numbers, and then returns a boolean indicating whether the element is an element of the input list. The function `sorted` takes in an input list and returns a boolean indicating whether it is sorted. Let's discuss them one by one:

Below is the definition of the `sorted` function:

```
-- (* Check that a list is sorted *)
val sorted: list nat -> Tot bool
let rec sorted l = match l with
    | []           -> true
    | [x]          -> true
    | x :: y :: xs -> (x <= y) && (sorted (y :: xs))
```

TODO: detailed explanations to be filled

Similarly, we show the definition of `elem` as follows:

```
-- (* Check that a natural number is an element of the natural number list *)
val elem: nat -> list nat -> Tot bool
let rec elem x l = match l with
  | []       -> false
  | hd :: tl -> hd = x || elem x tl
```

TODO: detailed explanations to be filled

With the defined `sorted` and `elem` lemmas, let's go back and look at how `insertionSort` is verified. We pattern match on the input list and separately discuss in the following two cases:

First, let's look at the base case, when we have an empty input list.

```
[] -> []
```

***Sorted Proof*** By definition of the `sorted` lemma, we know that `sort [] = true`. But both the input list and the result list is an empty list. Hence it's immediate that `sorted l1 sorted [] = sorted l2`.

***Permutation Proof*** Now let's discuss about the permutation proof `forall k. elem k l2 == elem k l1`. We want to show that every element in the input empty list is an element in the result list, which is also an empty list. It follows immediately from our definition of `elem` that `forall k.  elem k l1 = elem k [] = false = elem k [] = elem k l2`.

```
hd :: tl -> insert hd (insertion_sort tl)
```

Next, let's look at the proof when the input list contains at least one element.

TODO: details to be filled

(Second Draft reviews may skip to Summary) TODO: I will separately discuss the following proves as above.

```
val insert : x:nat -> l1:list nat{sorted l1} -> Tot (l2:list nat{sorted l2 /\
    (forall k. elem k l2 <==> k == x \/ elem k l1)})
let rec insert x l1 = match l1 with
  | []       -> [x]
  | hd :: tl ->
     if x <= hd then
       x :: l1
     else
       (* Solver implicitly uses the lemma: sorted_smaller hd (Cons.hd i_tl) tl
          *)
       hd :: (insert x tl)


(* A lemma to help Z3 *)
val smaller_hd_lemma: x:nat
                   -> y:nat
                   -> l:list nat
                   -> Lemma (requires (sorted (x::l) /\ elem y l))
                            (ensures (x <= y))
                            [SMTPat (sorted (x::l)); SMTPat (elem y l)]
let rec smaller_hd_lemma x y l = match l with
    | hd :: tl -> if hd = y then () else smaller_hd_lemma x y tl
```

#### 4.1.3.3 Summary

From a thorough comparison of the insertion sort proof in both Dependent Haskell and F*, we conclude that F* is comparably easy and relatively intuitive to program with. The complete verification in F* only takes less than 30 lines of code, but the same verification in Dependent Haskell takes almost 100 lines of code.

Below we summarize the key features of F* that contribute to its user-friendly interface for verification:

***Refinement Types*** With the support of refinement types, F* saves programmers from the burden of proof terms implementations. Instead, proving in F* feels a lot more like writing proofs by hand. Start from a function, such as the `insertionSort` above, we can directly encode the logic in the refinement predicate of its function signature and provide helper lemmas to complete the verification.

***Proof Automation with SMT Solver*** Recall from section 2.5.4 that F* achieves semi-automation through a combination of SMT solving using the Z3 solver and user-provided lemmas. Once we encode the theorems using refinement types, F* will generate verification conditions (VC) based on it and try to discharge the proof automatically through the Z3 SMT solver. Specifically in the sorted proof on `insert`, the transitivity of the `<=` operator is automatically proved by Z3. When there is a gap in the verification beyond the ability of Z3, for example in the second case of `insert`, users can provide a lemma to fill in the gap. F* provides the `SMTPat` keyword that supports pattern matching on logical predicates, so Z3 can automatically carry out the provided helper lemma to

complete the proof.

***Intuitive programming with Syntactic Sugar*** Finally, as discussed in section 2.2 that a refinement type, of the form `x:t{phi(x)}`, provides an additional logical predicates on the term of type `t` [3]. Internally, the predicate `phi(x)` is a type dependent on the term that will be transformed into VCs by the F* type system. We discussed briefly earlier this section that F* provides a lot of propositional connectives, such as `==`, $\wedge$, $\vee$, $\sim$, `<==`, `==>`, `<==>` etc, to hide the details of type construction. These syntactic sugar allows programmers to use boolean functions directly in the predicate instead of messing with types, and further improves the experience of theorem proving in F*.

On the other hand, Dependent Haskell requires encoding for all proof terms using dependent types. Programmers also need to provide every single step of the proof due to a lack of automation. Programming in Dependent Haskell requires a deeper understanding of type theory and can be tedious and hard.

Throughout our comparison on proving some more complicated theorems, however, we start to find that F*'s interface get hard to use very quickly while that of Dependent Haskell almost remains the same. Future work still need to be done to identify the "watershed" point. We believe that F* does make the relatively easy proofs easy, but the proofs on hard properties still remain hard. In general, we conclude that F* has a very good abstraction over its type system that provides an intuitive and friendly interface for theorem proving.

## 4.2 Approaches to Potentially Non-terminating Functions

In the second part of our comparison, we focus on potentially non-terminating functions. Truly divergent functions don't have many interesting properties that worth verify, so are not of our major concern. The reason why we are interested in the approaches towards non-terminating functions is that termination is generally hard to prove. When a language cannot prove some logically valid properties, its type system is incomplete. As a result, if a type system of a verification-oriented language can't verify properties on potentially non-terminating functions, then the promise of formal verification is weakened. On the other hand, formal verification through type system design aims to check properties at compile-time. However, without proper termination checking, erroneous proofs can pass type check at compile time and require further validation at run time. We define these type systems to be weak. Both incomplete and weak systems undercut the motivation of a verification-oriented programming language, but achieving both in full is a real theoretical challenge. Therefore, we want to compare and study the trade-offs between these two approaches towards non-termination.

We start from exploring the Peano division example (whose complete implementations are provided in Appendix). Peano naturals are a simple way to encode natural numbers with the `Zero` base value and the `Succ` recursive successor, each time incrementing the value by one. For example, natural number 0 is expressed as `Zero`, 1 is expressed as `Succ Zero` and 2 is expressed as `Succ (Succ Zero)` etc. Using the Peano division example,

we want to prove the property that the quotient of a product of two natural numbers $m, n$ and $n$ is equal to $m$, i.e. $m * n/n = m$. As an educational example to simulate non-termination, we allow zero divisors in our definition of `division` and try to prove the property when the divisors are positive, using the divergent division definition.

### 4.2.1 Dependent Haskell might be Complete towards Potentially Divergent Functions

Dependent Haskell doesn't require termination checking. With the help from type families and singletons, Dependent Haskell can successfully verify the Peano division property.

We first define the Peano natural numbers. The definition of the `Peano` ADT is similar to that of `Nat` as introduced in section 2.4.1.

```
data Peano = Zero | Succ Peano
```

Note that the Peano natural numbers are nonnegative. Then, we define division as follows:

```
-- recursive definition of integer division (/)
type family (m :: Peano) / (n :: Peano) :: Peano where
  Zero / _ = Zero
  m / n    = (Succ Zero) + (m - n) / n

-- (+), (-), and (*) are defined similarly
```

When dividing Peano natural number `a` by `b`, we first subtract the divisor `b` from the dividend `a`, then recursively divide the difference by `b`, and finally increment the quotient by `1`. Notice that our Consider the example calculating the quotient of `3` and `2`. Unwrapping the definition of division and subtraction, we get `3/2 = 1 + (3-2)/2 = 1 + (2-1)/2 = 1 + (1-0)/2 = 1 + 1/2 = 1`, where we simplify the peano naturals with natural numbers. (TODO: Update the division definition above in progress.) As long as divisor `b` is nonzero, we are recursively calling division on a smaller dividend, so division will terminate. As we want to explore properties on non-terminating functions, we deliberately give `b` the type `Peano`, including the case when `b = Zero`.

Other operations `+`, `-` and `*` are defined similarly. We define all four operations in the type level using type families so that they can be used in the type signature of the proof `peanoDivisionPf`.

Finally, we prove the property `(m * n)/n = m` for positive `n` in `peanoDivisionPf`. Note that the property `(m * n)/n = m` is only valid when `n > 0`, so we restrain the divisor to nonzero. We can define, using the corresponding singletons for `Peano`, the positive divisor as `SPeano (Succ n)`, so the property becomes `(m * (Succ n)) / (Succ n) = m`. The type signature of `peanoDivisionPf` is shown below:

```
peanoDivisionPf :: SPeano m -> SPeano (Succ n) ->
  (m * (Succ n)) / (Succ n) :~: m
```

The actual implementation of `peanoDivisionPf` is not very important here, though interested readers can go through the complete proof in Appendix .2.1. Similar to all verification examples introduced before, the proof of theorem `(m * (Succ n))/(Succ n):~:m` is carried out inductively. We pattern match on the dividend, then unwrap the definitions of each operator, apply a list of predefined lemmas, and recursively use the inductive hypothesis until we reach the property we want to show.

Dependent Haskell continues as Haskell to be a partial language, without compile-time termination checking. It allows programmers to define divergent functions like division, and tries to evaluate them in the function signature. If the theorem is valid, such as the property `(m * (Succ n))/(Succ n):~:m` we want to prove in our peano division example, then the evaluation on the divergent function will terminate with a sound, or correct, proof.

In summary, Dependent Haskell doesn't require any level of termination checking at compile time, so divergent functions such as division by zero can still be reasoned easily and used to prove the Peano division property valid in logic. We still need more examples to determine whether Dependent Haskell is complete.

### 4.2.2 F* is Incomplete towards Potentially Divergent Functions

F*, based on Hoare Logic, doesn't support extrinsic proofs on non-terminating functions. Using Hoare Logic, F* encodes the transition from one program state to another with Hoare triples. Hoare triple is of the form `P C Q`, where `P` and `Q` are called assertions in the predicate logic and denote the precondition and postcondition correspondingly. `C` is the command that is being executed. Hoare logic is a mature way to reason about program correctness, but is only relatively complete towards divergence, as it cannot derive the logically valid specifications if the execution `C` cannot be proved to terminate. As a result, F* only allows proving divergent programs intrinsically through type enrichment, as shown in the example of Peano Division.

First, we define the `peano` GADT in F* as follows:

```
type peano: Type =
 | Zero : peano
 | Succ : peano -> peano
```

We will present our explorations on the Peano division proof in the following two sections. Section 4.2.2.1 will focus on the failed attempt to prove in the extrinsic style. Section 4.2.2.2 will focus on a special case that could get around the limitation shown in 4.2.2.1 but only specifically for examples similar to Peano division.

#### 4.2.2.1 A Failed Attempt to Prove Extrinsically

To begin with, we try to prove the theorem `(m * n) / n = m` using the following definition of `division`:

```
val division: a:peano -> b:peano -> Dv (c:peano)
```

To explore whether F* has the ability to prove properties on potentially non-terminating functions, we again deliberately define a divergent `division` function. By definition of the `peano` type, the divisor `b` could be zero. Hence this `division` function defined above may not terminate and we annotate its effect to be `Dv`. The function signature for our peano division proof is shown below:

```
val peanoDivPf: m:peano -> n:peano{toNat n > 0} -> Lemma (ensures (division
    (mult m n) n == m))
```

Compiling the above code results in an error. The error states that a ghost expression is expected but the type system gets an expression with a divergent effect. Ghost is another effect predefined in F*'s type system that encapsulates computationally irrelevant code [17]. Let us first talk about why F* system expects a ghost expression. The return type starting with the `Lemma` keyword is a syntactic sugar F* provides for a more complex type `GTot (u:unitdivision (mult m n) n == m)`. Internally, a lemma is represented as a ghost total function that always returns a unit value. A unit value is a value that holds no information. As such, lemmas always return the same meaningless value and all we care about are the properties it carries. These properties can be specified in the refinement predicate of the unit value, or after the `ensures` clause using the syntactic sugar, and are verified at compile time.

Next we discuss the actual error and why the divergent `division` function mismatches with the expected ghost expression. Using the `Lemma` keyword, we express the theorem `division (mult m n) n == m` as a postcondition of `peanoDivPf` to be verified. The error occurs because we are trying to apply a divergent function with effect `Dv` to the postcondition that expects a computationally irrelevant total expression of effect `GTot` (stands for ghost total). F* designs its effect system to categorize the divergent `Dv` effect differently from all other effects [17]. The core difference is that the `Dv` effect is interpreted in a partial correctness setting. Recall our discussion in the beginning of this section on Hoare Logic. Given the precondition, command and postcondition in a Hoare triple, the logic can only validate the postcondition when each assertion in the precondition terminates [1]. Hence, with partial correctness specifications, we cannot decide a general program invariant to perform verifications. As a result, even though F* type system accepts potentially divergent functions, it refuses to evaluate them during type checking.

In summary, F* fails to directly prove lemmas on the divergent division function extrinsically. Even though this specific example might not be interesting as in reality we almost never define the division function with a zero divisor, the seemingly lack of support to prove divergent functions is considered a disadvantage. Acknowledging the theoretical limitations in verification, we are then curious to know whether it provides any workarounds, such as programming flags, to enable this feature.

#### 4.2.2.2 A Workaround to Prove Intrinsically

Let's start with a discussion on a valid Peano division proof in F*. As we clearly know the reason of nontermination in this educational example, we first provide a refinement condition to the divisor `b` to ensure that it's nonzero. The refined definition of `division_terminating` is shown below:

```
val division_terminating: a:peano -> b:peano{toNat b > 0} -> Tot (c:peano)
(decreases (toNat a))
```

There are two concepts worth mentioning in this `division_terminating` definition. First, `toNat` is a total function that takes in a `peano` GADT and returns the corresponding natural number, i.e. `toNat Zero = 0` and `toNat (Succ Zero) = 1` etc. Second, recall from section 2.5.5 that the F* type system depends on a well-founded partial order to perform termination checking. Specifically for the `division_terminating` function above, the logic is too complicated for F* to automatically deduce a metric from its four default termination metrics. As a result, we need to explicitly provide our own metric using the `decreases` keyword. Remember from the previous section on Dependent Haskell that division is defined as `1 + (a - b) / b`. We can quickly find that given a nonzero divisor `b`, the dividend `a` is decreasing at each recursive step. Hence we provide the termination metric as `decreases (toNat a)` following the main type signature.

With the above refinement and our explicit termination metric, the division function can be proved to terminate. Hence, our proof can then be successfully discharged with the following function signature:

```
val peanoDivPf: m:peano -> n:peano{toNat n > 0} -> Lemma (ensures
    (division_terminating (mult m n) n == m))
```

##### *The Workaround*

Then we return back to the divergent division definition and move on to discuss a Peano division proof that is successfully verified.

As F* does not support extrinsic proofs on non-terminating computations, we shift our focus to explore the possibility to prove intrinsically. However, the lemma `(a * b) / b = a` we want to prove is not a condition on any single function, but a combination of both multiplication and division. As such, we cannot simply refine the type of a function. Dependent types also failed here since the logic is too complicated to be encoded only using types (it might be possible but really hard in practice). In fact, the workaround we come up with, with some help from Tahina Ramananandro from the F* team, is not a flag-based solution. The solution is a little hacky and might be tricky to understand on a first read.

Specifically, our workaround to prove the lemma is by explicitly establishing the logical connection between the divergent function `division` and the total function `division_terminating` as introduced above. In the divergent `division` definition, we can intrinsically ensure that for any precondition, as long as the divisor for `division` is nonzero, the result of

`division` is always equivalent to that of `division_terminating`. It follows that we can complete the proof by applying the total function `division_terminating` in the postcondition of the proof `peanoDivPf`.

Question: Would a picture here denoting the relationship between these functions helpful to readers?

Below we show the modified divergent `division` definition as well as the type signature of `peanoDivPf`:

```
val division: a:peano -> b:peano -> Dv (c:peano{toNat b > 0 ==> c =
    div_terminating a b})

val peanoDivPf: m:peano -> n:peano{toNat n > 0} -> Lemma (ensures
    (division_terminating (mult m n) n == m))
```

Notice that this `peanoDivPf` above has exactly the same signature as that of the valid Peano division proof introduced at the beginning. Readers might wonder that the proof is verifying the terminating function `division_terminating`, not the `division` function we actually want to verify. In fact, the key point of the workaround is to prove intrinsically the equivalence of `division_terminating` and `division` under some logical conditions. Consider the modified divergent `division` function and notice the change we make on the function return type. Recall that the symbol `==>` stands for logical implies. We want to prove, intrinsically through type enrichment, that if the divisor `b` is greater than zero, then the result of the divergent `division` function is the same as that of the terminating `division_terminating` function. Therefore, given that we have a proof on the terminating `division_terminating` function, we indirectly have a proof for the `division` function we want to verify.

Our readers might be confused about this approach, as it really seems like a tautology. In fact, this workaround is a specific edge case that is not generally applicable. Though the introduction of the intermediary total function `division_terminating` completes the proof, this workaround approach inevitably requires programmers to know the exact reason of non-termination. Yet in reality, proving that functions terminate is not always feasible. Therefore we believe that F* is incomplete towards verifying potentially divergent functions.

## 4.3 Summary

TODO: to be continued

# 5 Conclusion

TODO: to be continued

# 6 Future Work

TODO: to be filled

# 7 Related Work and Contributions

TODO: to be filled

## 7.1 Language Comparison

There are many programming languages in use today and new ones are created every year. Various work has been done to compare programming languages. Through comparison, researchers find advantages and limitations in each and evaluate to propose insights for future research [9].

In the realm of programming languages, there are general-purposed programming languages like Pascal, Haskell and C/C++, interactive proof assistants like Coq, Agda and Isabelle/HOL, and SMT-based semi-automated program verification tools like Dafny, Vampire [12] and WhyML. Wiedijk compared a list of proof assistants on mathematical theorems, focusing on their strength of logic and level of automation [18] [19]. Feuer and Gehani edited papers comparing and assessing general purposed languages Ada, C and Pascal [9]. Filliâtre compared Vampire and Dafny in proving steps of the progress proof in Simply Typed Lambda Calculus [16] [10].

(to be filled: summarize what they did and found)

As the needs grow for general purpose yet verification oriented programming languages, four functional programming languages have been recently invented, namely Dependent Haskell, Liquid Haskell, F* and Idris. As an ongoing field of research, to our best knowledge, these four communities generally conduct research separately and have rarely explored each other's approaches in-depth.

At the 45th ACM POPL Student Research Competition, Xu and Zhang presented their collaborated work on the comparison among three programing verification techniques in Dependent Haskell, Liquid Haskell and F* [20]. As an exploratory research, they found promising aspects in each language and proposed more detailed directions for further comparisons. Our thesis provide an in-depth comparison between Dependent Haskell and F*, with a focus on their user interface design and different verification approaches towards potentially non-terminating functions. Through in-depth assessments of the verification techniques in both languages, we aim to find the more desirable approach and to recommend future directions for research.

## 7.2 Dependent Haskell

to be filled

### 7.2.1 Functional Dependency

to be filled

### 7.2.2 Constrained Type Families

to be filled

### 7.2.3 Promoting Functions to Type Families in Haskell

to be filled

## 7.3 F*

to be filled

### 7.3.1 Relative Completeness in Hoare Logic

to be filled

### 7.3.2 Dijkstra Monads

to be filled

### 7.3.3 Hereditary Substitutions

to be filled

## 7.4 The Zombie Language

to be filled (Details about how the Zombie Language attempts to verify divergent functions)

# 8 Glossary

TODO: to be filled (if time permits)

# Appendices

## .1 Insersion Sort

### .1.1 Dependent Haskell

```
{-# LANGUAGE IncoherentInstances, ConstraintKinds, TypeFamilies,
             TemplateHaskell, RankNTypes, ScopedTypeVariables, GADTs,
             TypeOperators, DataKinds, PolyKinds, MultiParamTypeClasses,
             FlexibleContexts, FlexibleInstances, UndecidableInstances #-}

module MyInsertionSort where

import Data.Kind (Type)
-- Mimics the Haskell Prelude, but with singleton types.
import Data.Singletons.Prelude
import Data.Singletons.SuppressUnusedWarnings
-- This module contains everything you need to derive your own singletons via
    Template Haskell.
import Data.Singletons.TH

-- Natural numbers, defined with singleton counterparts
$(singletons [d|
  data Nat = Zero | Succ Nat
  |])

-- A proof of the less-than-or-equal relation/constraint among naturals
data LeqProof :: Nat -> Nat -> Type where
  LeqZero :: LeqProof Zero a
  LeqSucc :: LeqProof a b -> LeqProof (Succ a) (Succ b)

-- A proof term asserting that a list of naturals is in ascending order
data NonDecProof :: [Nat] -> Type where
  AscEmpty :: NonDecProof '[]
  AscOne  :: NonDecProof '[n]
  AscCons :: LeqProof a b -> NonDecProof (b ': rest) -> NonDecProof (a ': b ':
      rest)

-- A proof term asserting that l2 is the list produced when x is inserted
    (anywhere) into list l1
data InsertionProof (x :: k) (l1 :: [k]) (l2 :: [k]) where
  -- Inserting x to the front of list l produces the list (x ': l)
  InsHere  :: InsertionProof x l (x ': l)
  -- If we get l2 after inserting x into l1, then we would get (y ': l2) when
      we insert x into (y ': l1)
  InsLater :: InsertionProof x l1 l2 -> InsertionProof x (y ': l1) (y ': l2)

-- A proof term asserting that l1 and l2 are permutations of each other
data PermutationProof (l1 :: [k]) (l2 :: [k]) where
  PermId  :: PermutationProof l l
  PermIns :: InsertionProof x l2 l2' -> PermutationProof l1 l2 ->
             PermutationProof (x ': l1) l2'

-- Here is the definition of functions about which we will be reasoning:
$(singletons [d|
  leq :: Nat -> Nat -> Bool
  leq Zero _        = True
```

```
  leq (Succ _) Zero = False
  leq (Succ a) (Succ b) = leq a b
  insert :: Nat -> [Nat] -> [Nat]
  insert n []    = [n]
  insert n (h:t) = if leq n h then (n:h:t) else h:(insert n t)
  insertionSort :: [Nat] -> [Nat]
  insertionSort []    = []
  insertionSort (h:t) = insert h (insertionSort t)
  |])

-- A lemma that states if sLeq a b is STrue, then we have a proof for (a :<=: b)
-- This is necessary to convert from the boolean definition of <= to the
    corresponding proof term to use in the type level
sLeq_true__le :: (Leq a b ~True) => SNat a -> SNat b -> LeqProof a b
sLeq_true__le a b = case (a, b) of
  (SZero, _)           -> LeqZero
  (SSucc a', SSucc b') -> LeqSucc (sLeq_true__le a' b')

-- A lemma that states if sLeq a b is SFalse, then we have a proof for (b :<=:
    a)
sLeq_false__nle :: (Leq a b ~False) => SNat a -> SNat b -> LeqProof b a
sLeq_false__nle a b = case (a, b) of
  (SSucc _, SZero)     -> LeqZero
  (SSucc a', SSucc b') -> LeqSucc (sLeq_false__nle a' b')

-- A lemma that states that inserting into an ascending list produces an
    ascending list
insert_ascending :: forall n lst. SNat n -> SList lst -> NonDecProof lst ->
    NonDecProof (Insert n lst)
insert_ascending n lst ascPf_lst = case ascPf_lst of
  AscEmpty               -> AscOne -- If lst is empty, then we're done
  AscOne                 -> case lst of -- If lst has one element...
    SCons h SNil -> case sLeq n h of -- then check if n is <= h
      STrue  -> AscCons (sLeq_true__le n h) AscOne -- if so, we're done
      SFalse -> AscCons (sLeq_false__nle n h) AscOne-- if not, we're done
  AscCons hLeqH2 ascPf_t -> case lst of -- Otherwise, if lst is more than one
      element...
    SCons h t -> case sLeq n h of -- then check if n is <= h
      STrue -> AscCons (sLeq_true__le n h) ascPf_lst -- if so, we're done
      SFalse -> case sLeq_false__nle n h of -- if not, things are harder...
        hLeqN -> case insert_ascending n t ascPf_t of
          ascPf_nt -> case t of
            SCons h2 _ -> case sLeq n h2 of
              STrue -> AscCons hLeqN ascPf_nt -- since we have h <= n and
                  t(h2:_) is ascending
              SFalse -> AscCons hLeqH2 ascPf_nt -- since we have h <= h2 and
                  t(h2:_) with n inserted is ascending

-- A lemma that states that the result of an insertion sort is in ascending
    order
nondec_pf :: SList lst -> NonDecProof (InsertionSort lst)
```

```
nondec_pf lst = case lst of
  SNil     -> AscEmpty -- if the list is empty, we're done
  -- otherwise, we recur to find that insertionSort on t produces an ascending
     list, and then we use the fact that inserting into an ascending list
     produces an ascending list
  SCons h t -> case nondec_pf t of
    ascPf_t -> insert_ascending h (sInsertionSort t) ascPf_t

-- A lemma that states that inserting n into lst produces a new list with n
   inserted into lst.
insert_insertion :: SNat n -> SList lst -> InsertionProof n lst (Insert n lst)
insert_insertion n lst =
  case lst of
    SNil    -> InsHere -- if lst is empty, we're done (by def of insert)
    SCons h t -> case sLeq n h of -- otherwise, is n <= h? (work with
       Singletons)
      STrue -> InsHere -- if so, we're done (by def of insert)
      SFalse -> InsLater (insert_insertion n t) -- otherwise, recur

-- A lemma that states that the result of an insertion sort is a permutation of
   its input
permutation_pf :: SList lst -> PermutationProof lst (InsertionSort lst)
permutation_pf lst = case lst of
  SNil     -> PermId -- if the list is empty, we're done
  -- otherwise, we wish to use PermIns. We must know that t is a permutation of
     the insertion sort of t and that inserting h into the insertion sort of t
     works correctly:
  SCons h t ->
    case insert_insertion h (sInsertionSort t) of
      insertionPf -> PermIns insertionPf (permutation_pf t)

-- A theorem that states that the insertion sort of a list is both ascending
   and a permutation of the original
insertionSort_pf :: SList lst -> (NonDecProof (InsertionSort lst),
                                  PermutationProof lst (InsertionSort lst))
insertionSort_pf slst = (nondec_pf slst,
                         permutation_pf slst)
```

## .2 Peano Division

### .2.1 Dependent Haskell

```
{-# LANGUAGE GADTs, TypeInType, ScopedTypeVariables, StandaloneDeriving,
             TypeFamilies, TypeOperators, AllowAmbiguousTypes, TypeApplications,
             UndecidableInstances, DataKinds #-}

module PeanoDivisionDH where

import Data.Kind ( Type )
import Data.Type.Equality ( (:~:)(..) )
import Prelude hiding ( div )
```

```
data Peano where
  Zero :: Peano
  Succ :: Peano -> Peano
deriving (Eq, Ord, Show)

type family (a :: Peano) + (b :: Peano) :: Peano where
  Zero + b     = b
  (Succ a) + b = Succ (a + b)
infix 6 +

type family (m :: Peano) - (n :: Peano) :: Peano where
  Zero - _            = Zero
  m - Zero            = m
  (Succ m) - (Succ n) = m - n
infix 6 -

type family (a :: Peano) * (b :: Peano) :: Peano where
  Zero * _     = Zero
  (Succ a) * b = a * b + b
infix 7 *

type family (a :: Peano) / (b :: Peano) :: Peano where
  Zero / _ = Zero
  m / n    = (Succ Zero) + (m - n) / n
infix 7 /

data SNat n where
  SZero :: SNat Zero
  SSucc :: SNat n -> SNat (Succ n)
deriving instance Show (SNat n)

plus :: SNat m -> SNat n -> SNat (m + n)
plus SZero n     = n
plus (SSucc m) n = SSucc (plus m n)

minus :: SNat m -> SNat n -> SNat (m - n)
minus SZero _              = SZero
minus m SZero              = m
minus (SSucc m) (SSucc n) = minus m n

mult :: SNat m -> SNat n -> SNat (m * n)
mult SZero _     = SZero
mult (SSucc m) n = plus (mult m n) n

plusZero :: SNat m -> m + Zero :~: m
plusZero SZero     = Refl
plusZero (SSucc m) = case (plusZero m) of Refl -> Refl

plusSucc :: SNat m -> SNat n -> m + (Succ n) :~: Succ (m + n)
plusSucc SZero _     = Refl
```

```
plusSucc (SSucc m) n = case (plusSucc m n) of Refl -> Refl

--   (m + (Succ n)) - (Succ n)
-- = Succ (m + n) - (Succ n), by plusSucc
-- = (m + n) - n, by definition of (-)
-- = m, by inductive hypothesis
plusMinus :: SNat m -> SNat n -> (m + n) - n :~: m
plusMinus m SZero     = case (plusZero m) of Refl -> Refl
plusMinus m (SSucc n) =
   case (plusSucc m n) of Refl
-> case (plusMinus m n) of Refl
-> Refl

--   (Succ m * Succ n) / Succ n
-- = (m * Succ n + Succ n) / Succ n, by definition of (*)
-- = Succ (m * Succ n + n) / Succ n, by plusSucc
-- = Succ Zero + (Succ (m * Succ n + n) - Succ n) / Succ n, by definition of (/)
-- = Succ Zero + ((Succ (m * Succ n) + n) - Succ n) / Succ n, by definition of
    (+)
-- = Succ (((Succ (m * Succ n) + n) - Succ n) / Succ n), by definition of (+)
-- = Succ (((m * Succ n) + Succ n - Succ n) / Succ n), by plusSucc (same as
    above, opposite direction)
-- = Succ ((m * Succ n) / Succ n), by plusMinus
-- = Succ m, by inductive hypothesis
peanoDivisionPf :: SNat m -> SNat (Succ n) -> (m * (Succ n)) / (Succ n) :~: m
peanoDivisionPf SZero _                    = Refl
peanoDivisionPf (SSucc m) succ_n@(SSucc n) =
   case plusMinus (mult m succ_n) succ_n of Refl
-> case plusSucc (mult m succ_n) n of Refl
-> case peanoDivisionPf m succ_n of Refl
-> Refl
```

### .2.2 F*

```
module PeanoDivision

type peano: Type =
  | Zero : peano
  | Succ : peano -> peano

val toNat: peano -> Tot nat
let rec toNat a = match a with
  | Zero    -> 0
  | Succ a' -> 1 + toNat a'

val max: int -> int -> Tot int
let max a b = if a > b then a else b

val plus: a:peano -> b:peano -> Tot (c:peano)
let rec plus a b = match a with
  | Zero    -> b
```

```
  | Succ a' -> Succ (plus a' b)

val minus: a:peano -> b:peano ->
  Tot (c:peano{toNat c = max (toNat a - toNat b) 0})
let rec minus a b = match a with
  | Zero    -> Zero
  | Succ a' -> match b with
    | Zero    -> Succ a'
    | Succ b' -> minus a' b'

val mult: a:peano -> b:peano -> Tot (c:peano)
let rec mult a b = match a with
  | Zero    -> Zero
  | Succ a' -> plus (mult a' b) b

val division_terminating: a:peano -> b:peano{toNat b > 0} -> Tot (c:peano)
(decreases (toNat a))
let rec division_terminating a b = match a with
  | Zero   -> Zero
  | Succ _ -> if (toNat a < toNat b) then Zero
              else plus (division_terminating (minus a b) b) (Succ Zero)

val division: a:peano -> b:peano -> Div (c:peano) (requires True) (ensures (fun
    y -> toNat b > 0 ==> y == div_terminating a b))
let rec division a b = match a with
 | Zero   -> Zero
 | Succ _ -> if (toNat a < toNat b) then Zero
             else let denom = (division (minus a b) b)
                  in plus denom (Succ Zero)

val plusZero: m:peano -> Lemma (ensures plus m Zero = m)
let rec plusZero m = match m with
  | Zero   -> ()
  | Succ m -> plusZero m

val plusSucc: m:peano -> n:peano -> Lemma (ensures plus m (Succ n) = Succ (plus
    m n))
let rec plusSucc m n = match m with
  | Zero   -> ()
  | Succ m -> plusSucc m n

val plusMinus: m:peano -> n:peano -> Lemma
(requires True)
(ensures minus (plus m n) n = m)
let rec plusMinus m n = match n with
  | Zero   -> plusZero m
  (*
    minus (plus a (Succ b')) (Succ b')
  = minus (Succ (plus a b')) (Succ b') by plusSucc a b'
  = minus (plus a b') b'
  = a by minusPlus a b'
```

```
  *)
  | Succ n -> match plusSucc m n with () -> plusMinus m n

val peanoDivPf: m:peano -> n:peano{toNat n > 0} -> Lemma
(requires True)
(ensures (div (mult m n) n == m))
let rec peanoDivPf m n = match m with
  (*
    div (mult Zero b) b = div Zero b = Zero
  *)
  | Zero   -> ()
  (*
    div (mult (Succ a') b) b
  = div (plus (mult a' b) b) b
  = plus (div (minus (plus (mult a' b) b) b) b) (Succ Zero)
  = plus (div (mult a' b) b) (Succ Zero) by plusMinus (mult a' b) b
  = plus a' (Succ Zero) by peanoDivPf a' b
  = Succ (plus a' Zero) by plusSucc a' Zero
  = Succ a' by plusZero a'
  *)
  | Succ a' -> plusMinus (mult a' b) b;
               peanoDivPf a' b;
               plusSucc a' Zero;
               plusZero a'
```

# References


[1] Relative Completeness. https://www.cs.cornell.edu/courses/cs4110/2010fa/lectures/lecture09.pdf. Accessed: 2018-05-15.

[2] The 2017 Top Programming Languages. https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages. Accessed: 2018-05-15.

[3] Verified programming in F* - A Tutorial. https://www.fstar-lang.org/tutorial/tutorial.html#sec-the-st-effect. Accessed: 2018-05-15.

[4] John N Buxton and Brian Randell. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

[5] Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[7] Richard A. Eisenberg. *Dependent Types in Haskell: Theory and Practice*. PhD thesis, University of Pennsylvania, 2016.

[8] Richard A. Eisenberg and Stephanie Weirich. Dependently Typed Programming with Singletons. In *ACM SIGPLAN Haskell Symposium*, 2012.

[9] Narain Gehani. *Comparing and Assessing Programming Languages: Ada, C, and Pascal*. Prentice Hall, 1984.

[10] Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. Automating Proof Steps of Progress Proofs: Comparing Vampire and Dafny. In *Vampire@ IJCAR*, pages 33–45, 2016.

[11] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[12] Laura Kovács and Andrei Voronkov. First-order Theorem Proving and Vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[13] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'16, pages 348–370. Springer Berlin Heidelberg, 2010.

[14] K Rustan M Leino. Automating Induction with an SMT solver. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 315–331. Springer, 2012.

[15] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[16] Benjamin C Pierce. *Types and Programming Languages*. MIT press, 2002.

[17] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-monadic Effects in F$^\star$. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16. ACM, 2016.

[18] Freek Wiedijk. Comparing Mathematical Provers. In *International Conference on Mathematical Knowledge Management*, pages 188–202. Springer, 2003.

[19] Freek Wiedijk. *The Seventeen Provers of the World*, volume 3600. Springer, 2006.

[20] Rachel Xu and Xinyue Zhang. Comparisons among three programming verification techniques in Dependent Haskell, Liquid Haskell and F*. In *Student Research Competition at the Principals of Programming Languages*, POPL '18. ACM, 2018.

[21] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.